The Three Dimensions of Scalable Machine Learning

Reza Zadeh

@Reza_Zadeh | http://reza-zadeh.com

Outline

Data Flow Engines and Spark

The Three Dimensions of Machine Learning

Matrix Computations

MLlib + {Streaming, GraphX, SQL}

Future of MLlib

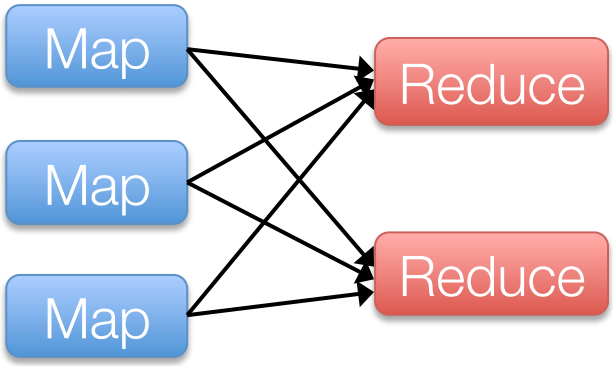
Data Flow Models

Restrict the programming interface so that the system can do more automatically

Express jobs as graphs of high-level operators

- » System picks how to split each operator into tasks and where to run each task
- » Run parts twice fault recovery

Biggest example: MapReduce



Spark Computing Engine

Extends a programming language with a distributed collection data-structure

» “Resilient distributed datasets” (RDD)

Open source at Apache

» Most active community in big data, with 50+ companies contributing

Clean APIs in Java, Scala, Python

Community: SparkR, being released in 1.4!

Key Idea

Resilient Distributed Datasets (RDDs)

- » Collections of objects across a cluster with user controlled partitioning & storage (memory, disk, ...)
- » Built via parallel transformations (map, filter, …)
- » The world only lets you make make RDDs such that they can be:

Automatically rebuilt on failure

Resilient Distributed Datasets (RDDs)

Main idea: Resilient Distributed Datasets

» Immutable collections of objects, spread across cluster

» Statically typed: RDD[T] has objects of type T

```
val sc = new SparkContext()
val lines = sc.textFile("log.txt")   // RDD[String]

// Transform using standard collection operations
val errors = lines.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split('\t')(2))

messages.saveAsTextFile("errors.txt")
```

→ lazily evaluated

→ kicks off a computation

MLlib: Available algorithms

classification: logistic regression, linear SVM, naïve Bayes, least squares, classification tree

regression: generalized linear models (GLMs), regression tree

collaborative filtering: alternating least squares (ALS), non-negative matrix factorization (NMF)

clustering: k-means||

decomposition: SVD, PCA

optimization: stochastic gradient descent, L-BFGS

The Three Dimensions

ML Objectives

Almost all machine learning objectives are optimized using this update

$$w \leftarrow w - \alpha \cdot \sum_{i=1}^{n} g(w; x_i, y_i)$$

Scaling

1) Data size

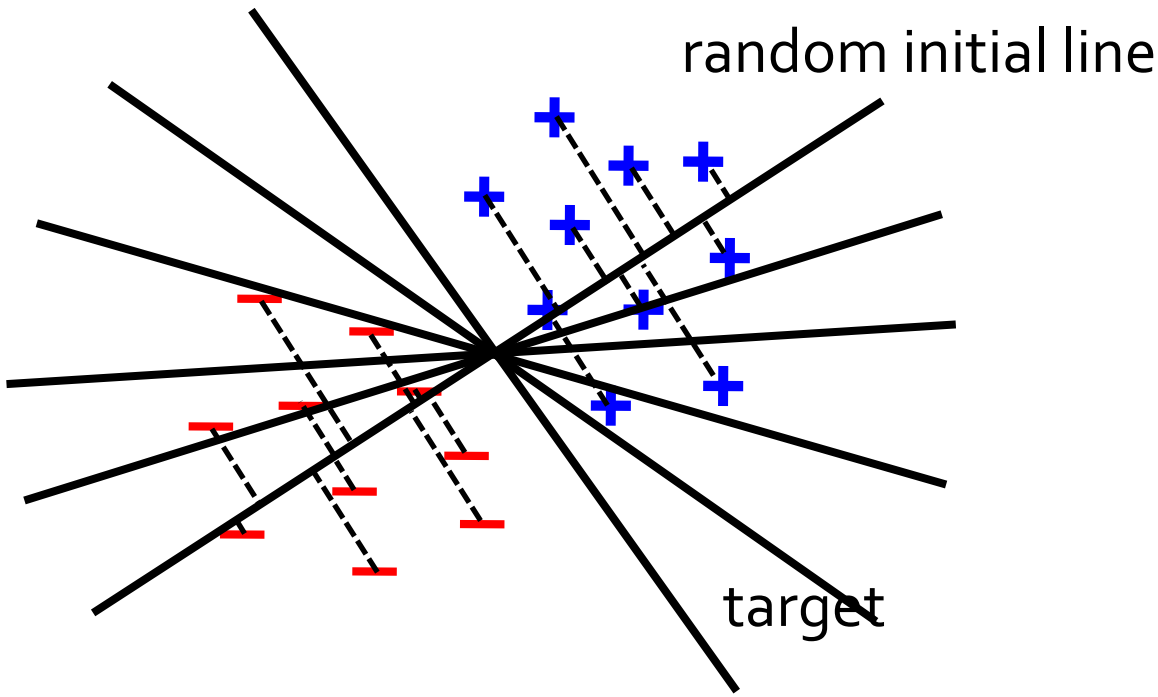
$$w \leftarrow w - \alpha \cdot \sum_{i=1}^{n} g(w; x_i, y_i)$$

2) Number of models

3) Model size

Logistic Regression

Goal: find best line separating two sets of points



Data Scaling

$$w \leftarrow w - \alpha \cdot \sum_{i=1}^{n} g(w; x_i, y_i)$$

```
val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.zeros(d)
for (i <- 1 to numIterations) {
  val gradient = points.map { p =>
    (1 / (1 + exp(-p.y * w.dot(p.x)) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= alpha * gradient
}
```

Separable Updates

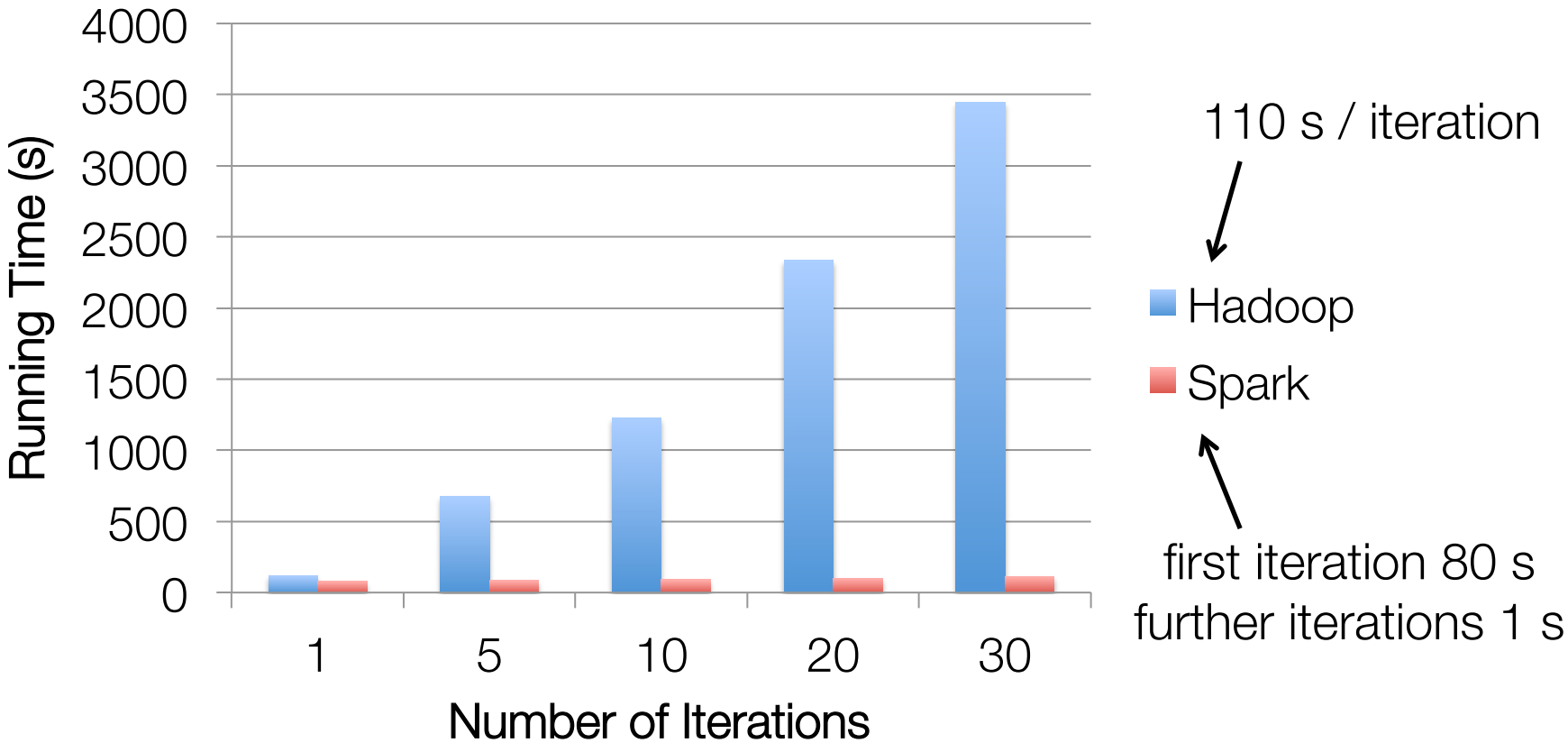
Can be generalized for

» Unconstrained optimization

» Smooth or non-smooth

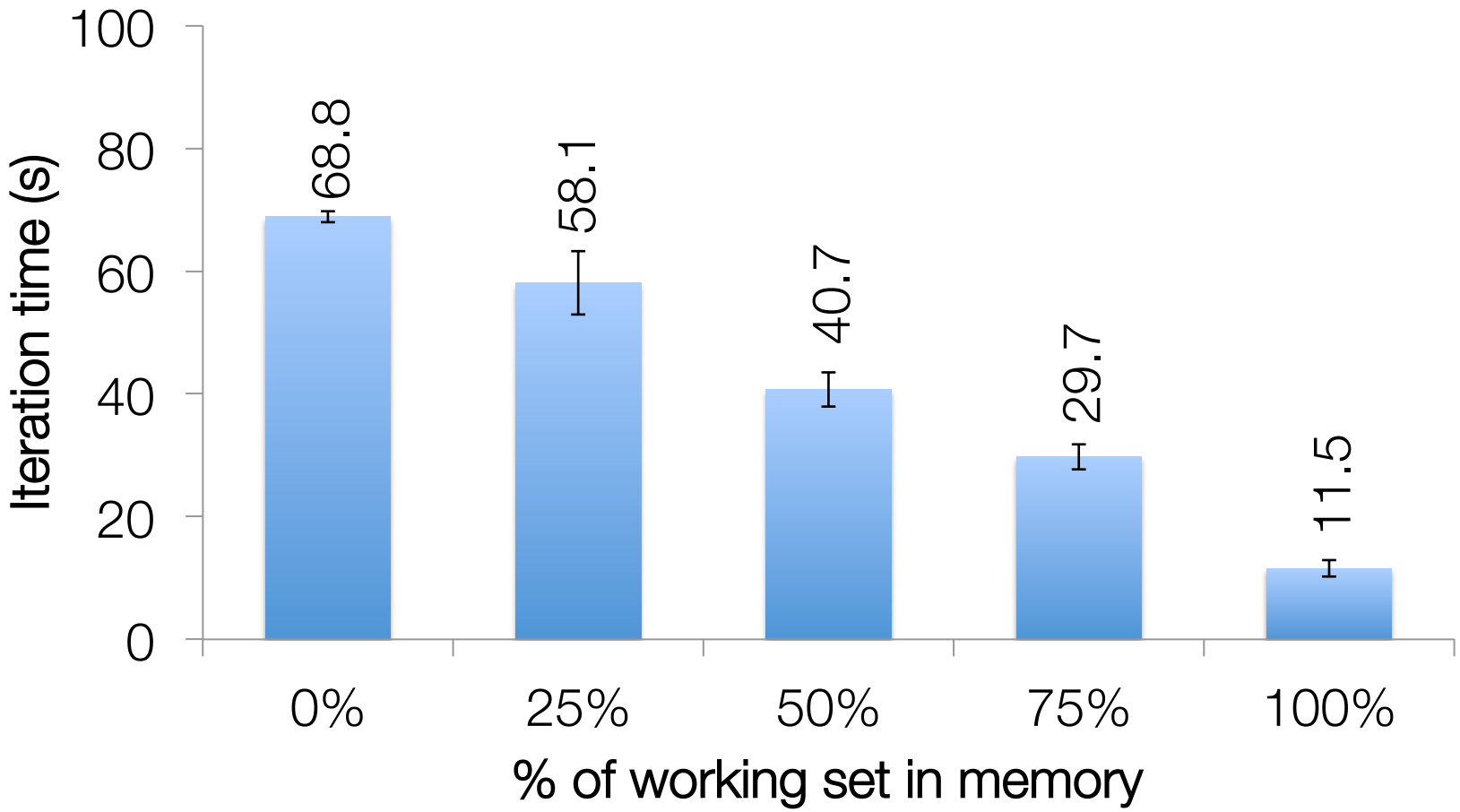
» LBFGS, Conjugate Gradient, Accelerated Gradient methods, …

Logistic Regression Results



100 GB of data on 50 m1.xlarge EC2 machines

Behavior with Less RAM



Lots of little models

Is embarrassingly parallel

Most of the work should be handled by data flow paradigm

ML pipelines does this

Hyper-parameter Tuning

```
// Build a parameter grid.
val paramGrid = new ParamGridBuilder()
  .addGrid(hashingTF.numFeatures, Array(10, 20, 40))
  .addGrid(lr.regParam, Array(0.01, 0.1, 1.0))
  .build()

// Set up cross-validation.
val cv = new CrossValidator()
  .setNumFolds(3)
  .setEstimator(pipeline)
  .setEstimatorParamMaps(paramGrid)
  .setEvaluator(new BinaryClassificationEvaluator)

// Fit a model with cross-validation.
val cvModel = cv.fit(trainingDataset)
```

Model Scaling

Linear models only need to compute the dot product of each example with model

Use a BlockMatrix to store data, use joins to compute dot products

Coming in 1.5

Model Scaling

Data joined with model (weight):

Optimization

At least two large classes of optimization problems humans can solve:

» Convex

» Spectral

Optimization Example: Spectral Program

Spark PageRank

Given directed graph, compute node importance. Two RDDs:

» Neighbors (a sparse graph/matrix)

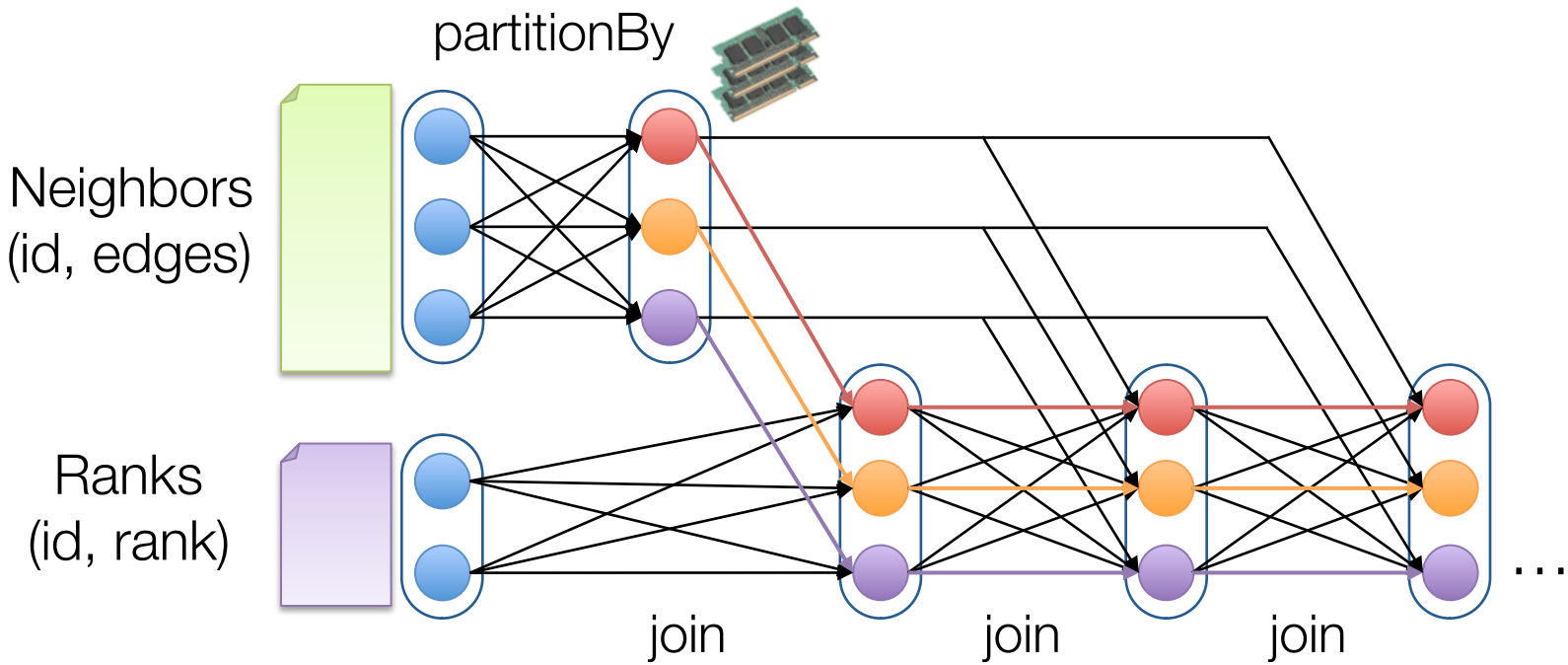
» Current guess (a vector)

Using cache(), keep neighbor list in RAM

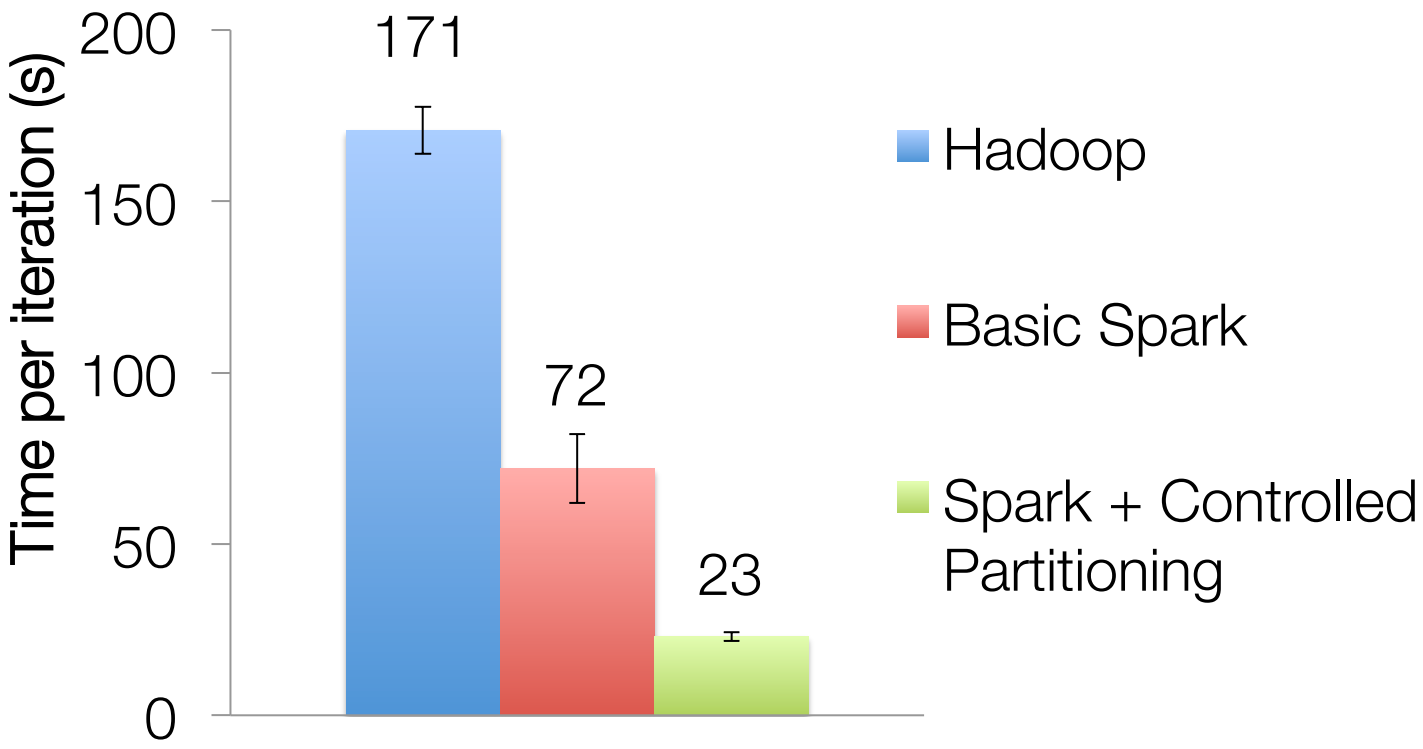
Spark PageRank

Using cache(), keep neighbor lists in RAM

Using partitioning, avoid repeated hashing



PageRank Results



Spark PageRank

Generalizes to Matrix Multiplication, opening many algorithms from Numerical Linear Algebra

Distributing Matrix Computations

Distributing Matrices

How to distribute a matrix across machines?

» By Entries (CoordinateMatrix)

» By Rows (RowMatrix)

» By Blocks (BlockMatrix) As of version 1.3

All of Linear Algebra to be rebuilt using these partitioning schemes

Distributing Matrices

Even the simplest operations require thinking about communication e.g. multiplication

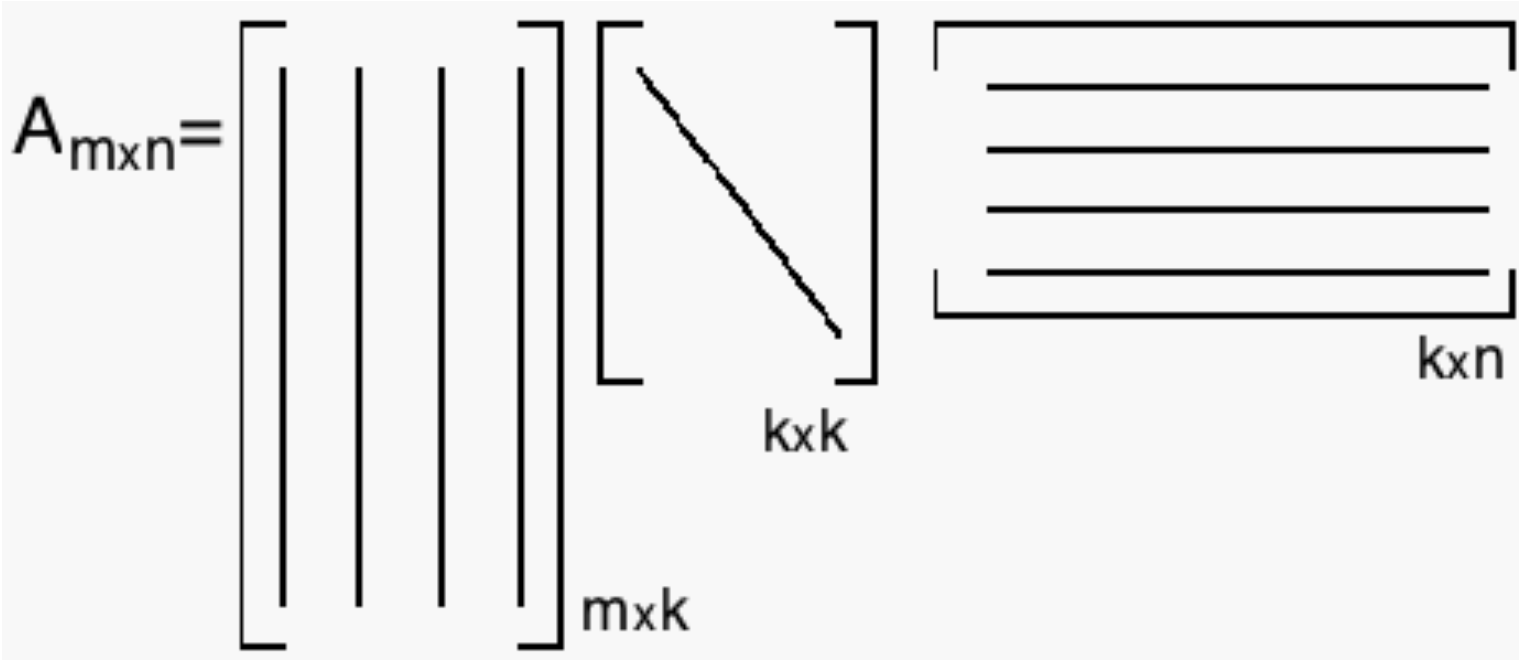
How many different matrix multiplies needed?

» At least one per pair of {Coordinate, Row, Block, LocalDense, LocalSparse} = 10

» More because multiplies not commutative

Singular Value Decomposition on Spark

Singular Value Decomposition



Singular Value Decomposition

Two cases

» Tall and Skinny

» Short and Fat (not really)

» Roughly Square

SVD method on RowMatrix takes care of which one to call.

Tall and Skinny SVD

- Given $m \times n$ matrix A, with $m \gg n$.
- We compute $A^T A$.
- $A^T A$ is $n \times n$, considerably smaller than A.
- $A^T A$ is dense.
- Holds dot products between all pairs of columns of A.

$$A = U\Sigma V^T \qquad\qquad A^T A = V\Sigma^2 V^T$$

Tall and Skinny SVD

$$A^T A = V \Sigma^2 V^T$$

Gets us V and the singular values

$$A = U \Sigma V^T$$

Gets us U by one matrix multiplication

Square SVD

ARPACK: Very mature Fortran77 package for computing eigenvalue decompositions

JNI interface available via netlib-java

Distributed using Spark – how?

Square SVD via ARPACK

Only interfaces with distributed matrix via matrix-vector multiplies

$$K_n = \begin{bmatrix} b & Ab & A^2b & \cdots & A^{n-1}b \end{bmatrix}$$

The result of matrix-vector multiply is small.

The multiplication can be distributed.

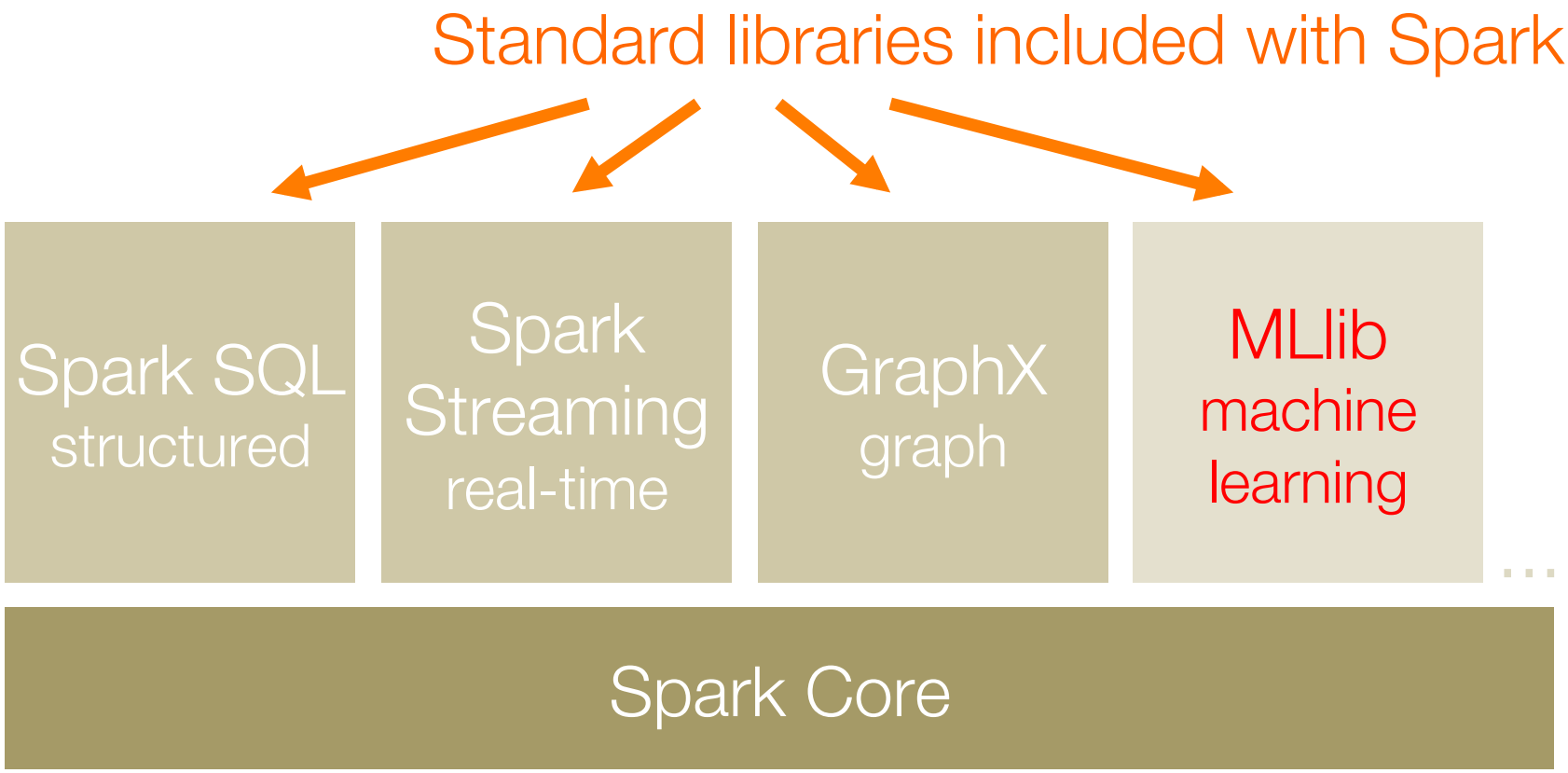
Square SVD

Matrix size	Number of nonzeros	Time per iteration (s)	Total time (s)
23,000,000 x 38,000	51,000,000	0.2	10
63,000,000 x 49,000	440,000,000	1	50
94,000,000 x 4,000	1,600,000,000	0.5	50

With 68 executors and 8GB memory in each, looking for the top 5 singular vectors

MLlib + {Streaming, GraphX, SQL}

A General Platform

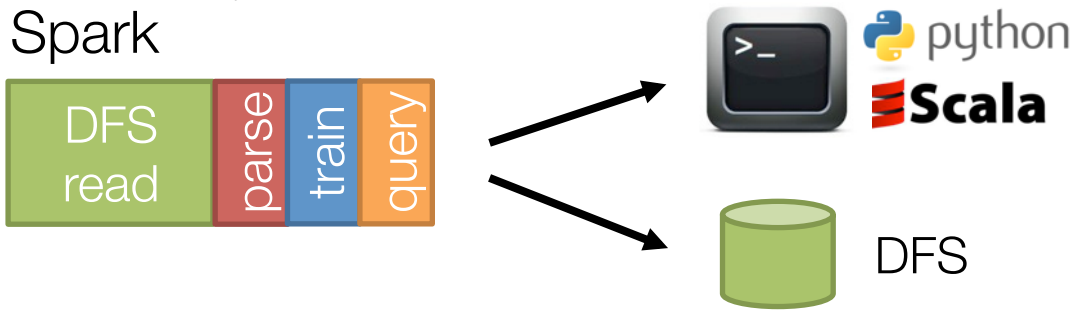


Benefit for Users

Same engine performs data extraction, model training and interactive queries

Separate engines

Spark



MLlib + Streaming

As of Spark 1.1, you can train linear models in a streaming fashion, k-means as of 1.2

Model weights are updated via SGD, thus amenable to streaming

More work needed for decision trees

MLlib + SQL

```
df = context.sql(“select latitude, longitude from tweets”)
model = pipeline.fit(df)
```

DataFrames in Spark 1.3! (March 2015)

Powerful coupled with new pipeline API

MLlib + GraphX

```
// assemble link graph
val graph = Graph(pages, links)
val pageRank: RDD[(Long, Double)] = graph.staticPageRank(10).vertices

// load page labels (spam or not) and content features
val labelAndFeatures: RDD[(Long, (Double, Seq((Int, Double)))] = ...
val training: RDD[LabeledPoint] =
  labelAndFeatures.join(pageRank).map {
    case (id, ((label, features), pageRank)) =>
      LabeledPoint(label, Vectors.sparse(features ++ (1000, pageRank))
}

// train a spam detector using logistic regression
val model = LogisticRegressionWithSGD.train(training)
```

Future of MLlib

Goals for next version

Tighter integration with DataFrame and spark.ml API

Accelerated gradient methods & Optimization interface

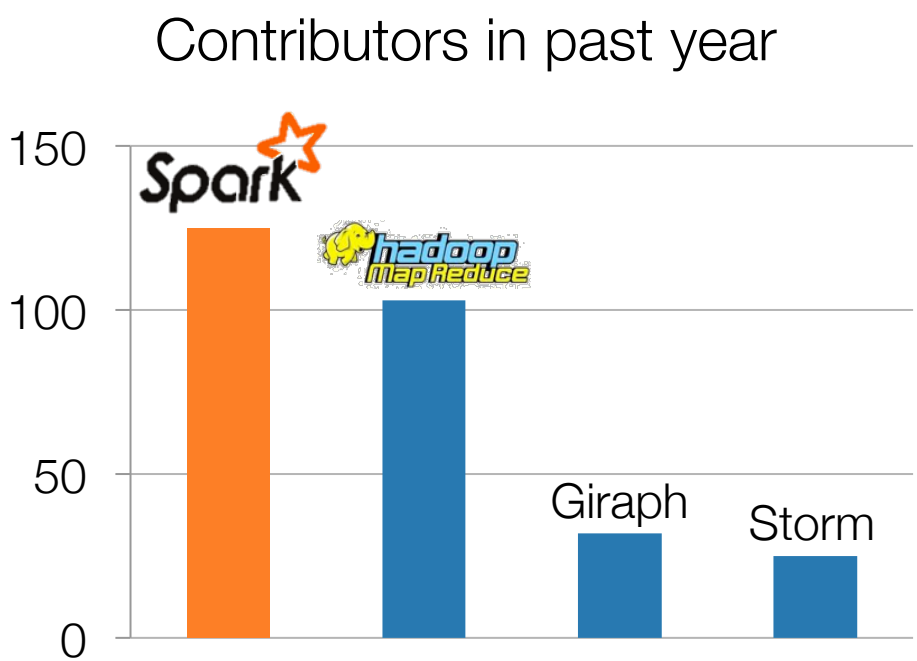
Model export: PMML (current export exists in Spark 1.3, but not PMML, which lacks distributed models)

Scaling: Model scaling (e.g. via Parameter Servers)

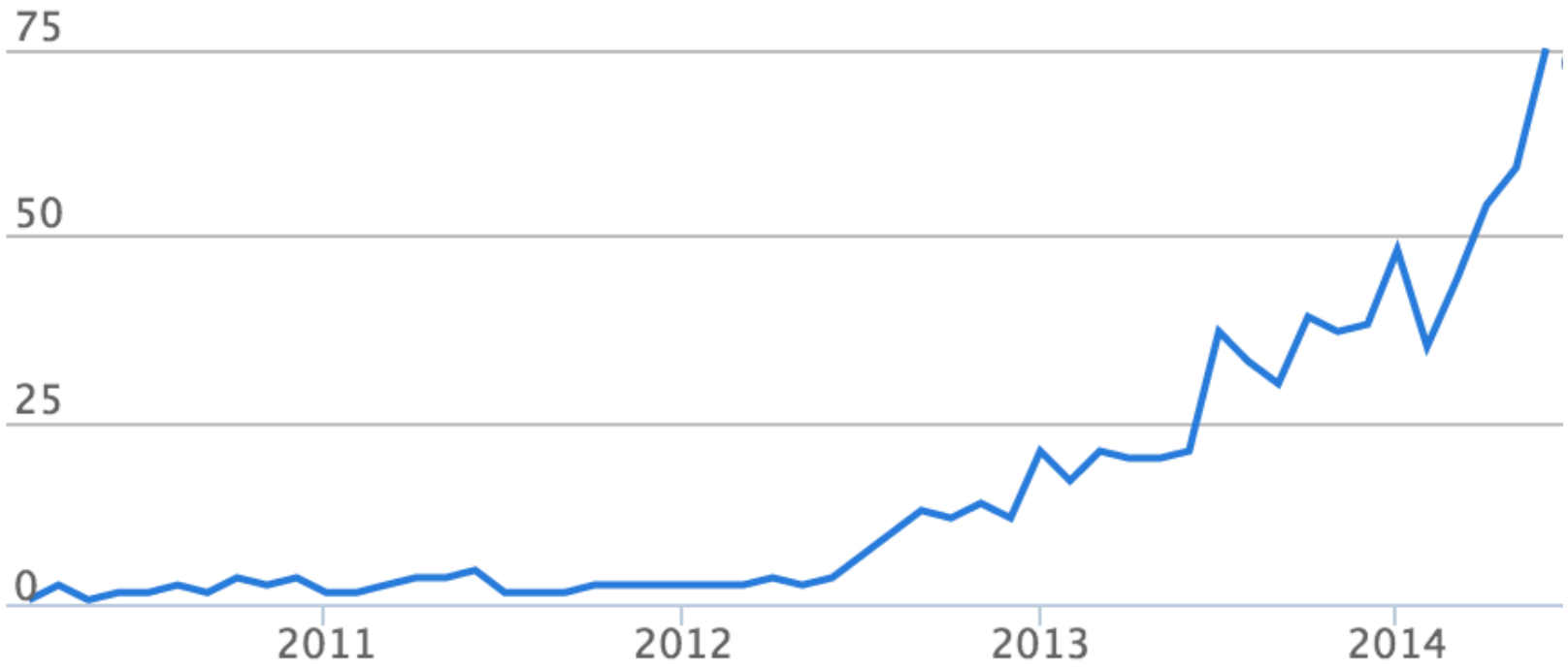
Spark Community

Most active open source community in big data

200+ developers, 50+ companies contributing



Continuing Growth



Contributors per month to Spark

source: ohloh.net

Spark and ML

Spark has all its roots in research, so we hope to keep incorporating new ideas!

Model Broadcast

$$w \leftarrow w - \alpha \cdot \sum_{i=1}^{n} g(w; x_i, y_i)$$

```
val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.zeros(d)
for (i <- 1 to numIterations) {
  val gradient = points.map { p =>
    (1 / (1 + exp(-p.y * w.dot(p.x)) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= alpha * gradient
}
```

Model Broadcast

$$w \leftarrow w - \alpha \cdot \sum_{i=1}^{n} g(w; x_i, y_i)$$

Call sc.broadcast

```
val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.zeros(d)
for (i <- 1 to numIterations) {
  val gradient = points.map { p =>
    (1 / (1 + exp(-p.y * w.dot(p.x)) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= alpha * gradient
}
```

Use via .value